Matthew Ruffell

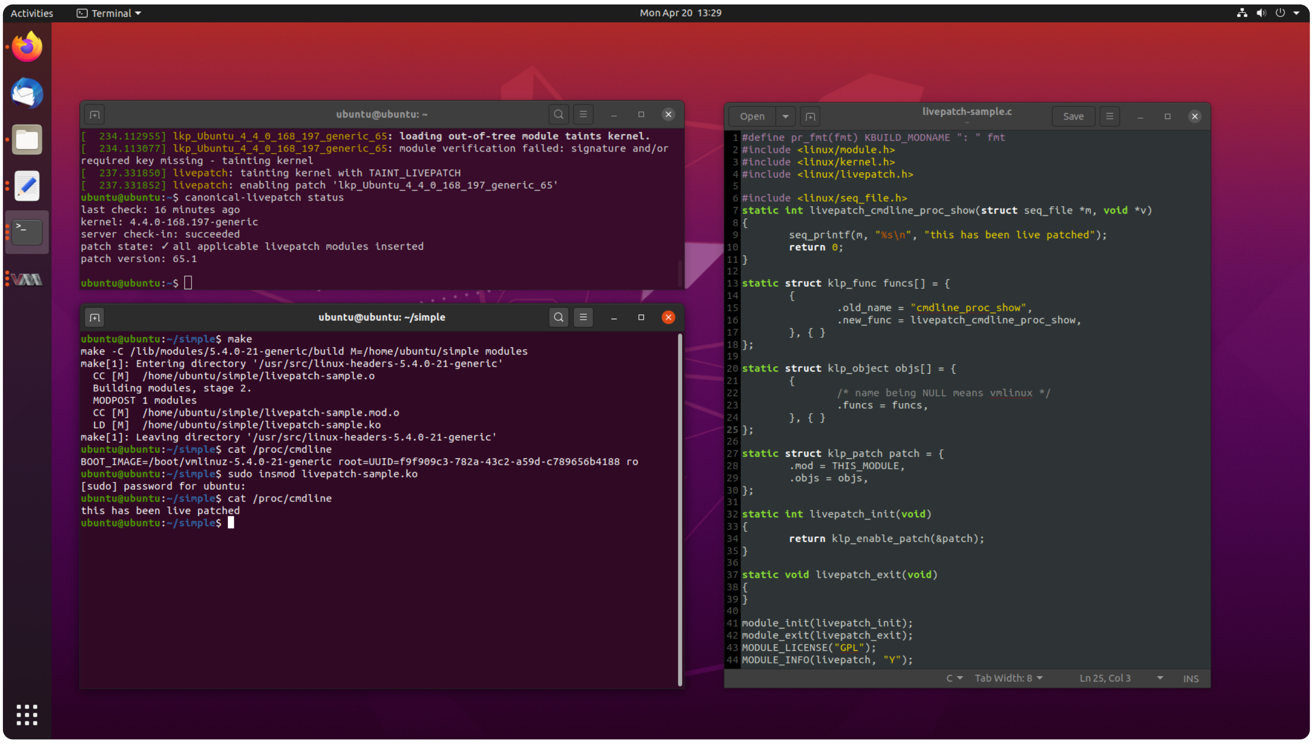
Everything You Wanted to Know About Kernel Livepatch in Ubuntu

20 Apr 2020 • Programming • Writeups

One of the more recent killer features implemented by most major Linux distros these days is the ability to patch the kernel while it is running, without the need for a reboot.

While this may sound like sorcery for some, this is a very real feature, called Livepatch. Livepatch uses ftrace in new and interesting ways, by patching in calls at the beginning of existing functions to new patched functions, delivered as kernel modules.

This lets you update and fix bugs on the fly, although its use is typically reserved for security critical fixes only.



The whole concept is extremely interesting, so today we will look into what Livepatch is, how it is implemented across several distros, we will write some Livepatches of our own, and look at how Livepatch works in Ubuntu for end users.

Why Do We Need Livepatch?

Working in Sustaining Engineering at Canonical, it is pretty common to see bug reports from machines which have very high uptimes, such as six to twelve months, or sometimes even longer.

These machines normally run important workloads which can't be interrupted for a reboot, since they might be a part of critical public infrastructure, or a busy build system. The Ubuntu Kernel Team typically releases a new updated kernel for each distribution release on a 3 week SRU cycle with additional updates always within a day of two of a new CVE being released.

Machines with important workloads aren't going to want to reboot every six months, let alone every three weeks for each new kernel release. Keeping these machines safe and up to date with security fixes is a must, and this is the motivation behind Livepatch.

What is Livepatch?

Livepatch is the ability for the kernel to change the flow of code execution from a broken or vulnerable function, to a new, fixed function during runtime.

In most cases, the new function is the exact same as the function it is replacing, but with minor changes, such as adding a check for null, or changing the order of some locks or adding a quick logic fix.

The code redirection is achieved with ftrace. ftrace is a tool which lets you trace kernel function calls, but it can also add and remove instructions from functions as well. A good example is kprobes, which can patch in blocks of code to existing functions, usually used to print debug values. kprobes are mostly ftrace based these days, which is important, since we don't want kprobes and Livepatch to clash and patch the same function at the same time, so ftrace controls function consistency.

Livepatch is implemented by compiling the new fixed function into a kernel module and loading it into the system. ftrace is then used to redirect calls from the old function to the new function in the kernel module. This process actually has to be done very carefully, and we will discuss it in the next section, when we cover different consistency models.

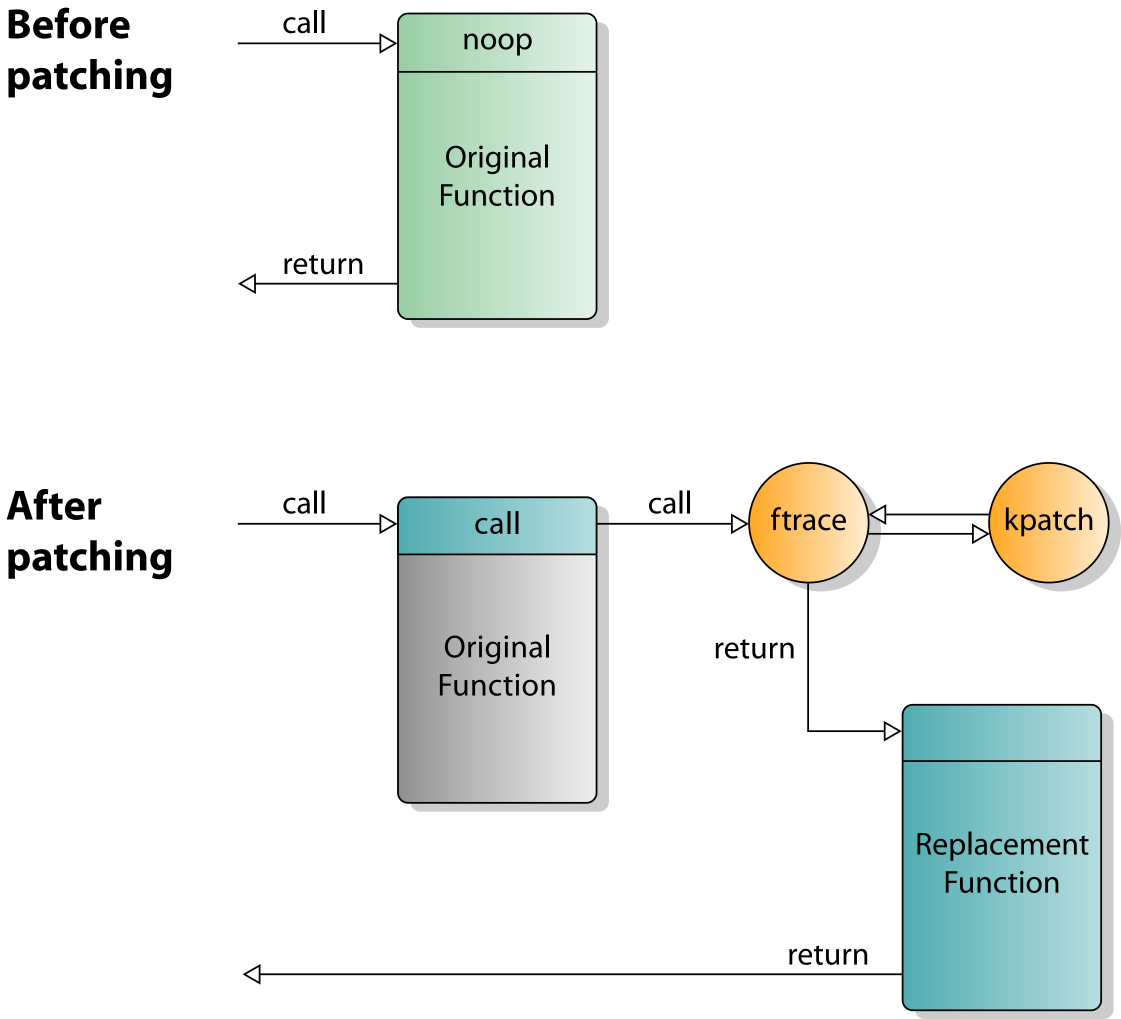
For the actual implementation, it is remarkably simple.

Have you ever disassembled a kernel function before and wondered why every kernel function begins with a full sized padded `nop` instruction?

For example, let's look at `sysrq_handle_crash()` , as seen in my previous article Beginning Kernel Crash Debugging on Ubuntu 18.10.

```
crash> dis sysrq_handle_crash
0xffffffff8c41d930 <sysrq_handle_crash>:       nop    DWORD PTR [rax+rax*1+0x0]
0xffffffff8c41d935 <sysrq_handle_crash+5>:     push   rbp
0xffffffff8c41d936 <sysrq_handle_crash+6>:     mov    DWORD PTR [rip+0x13637a8],0x1        # 0xffffffff8d7810e8
0xffffffff8c41d940 <sysrq_handle_crash+16>:    mov    rbp,rsp
0xffffffff8c41d943 <sysrq_handle_crash+19>:    sfence
0xffffffff8c41d946 <sysrq_handle_crash+22>:    mov    BYTE PTR ds:0x0,0x1
0xffffffff8c41d94e <sysrq_handle_crash+30>:    pop    rbp
0xffffffff8c41d94f <sysrq_handle_crash+31>:    ret
crash>
```

Well, what ftrace does is patch out the `nop` with a `call` which points towards the new function. If you look carefully, the `nop` is located before the function starts manipulating the stack, which means everything is consistent, and very elegant.



Credit and license for image

The above image demonstrates this behaviour very well. Now, this technique works great at a function level, where logic changes but data does not.

Limitations quickly arise within Livepatch when data changes are required. If a new member is needed to be added or removed from a struct implemented within the function or the file, these changes cannot be passed onto the Livepatched version, since you cannot modify data structures during runtime, as they may be in use by different tasks on different cpus. The same goes for changing the function signature, since the calling function would have to rearrange variables pushed on the stack. Livepatch is also limited to modifying functions which are traceable by ftrace, and not all kernel functions can be traced.

Because of these limitations, and the complexity that arises from consistency models which we will discuss about next, Livepatch is more of a temporary band-aid solution, reserved for fixing critical security issues until such a time comes when the host can be rebooted into a updated kernel.

Consistency Models and Varying Implementations

As mentioned in the previous section, the real complexity behind Livepatch is the decision making process required when ftrace actually performs the switch from the old function to the new function.

Say the changes to the new function are basic. Adding a null pointer check sort of basic. The semantics of the function itself haven't changed, and there is no existing state to manage. All we have to do then is check to see if any tasks are running which are using the old function. This can be done by examining the stack of sleeping tasks. If the function is not found in any of them, we can easily patch the change in.

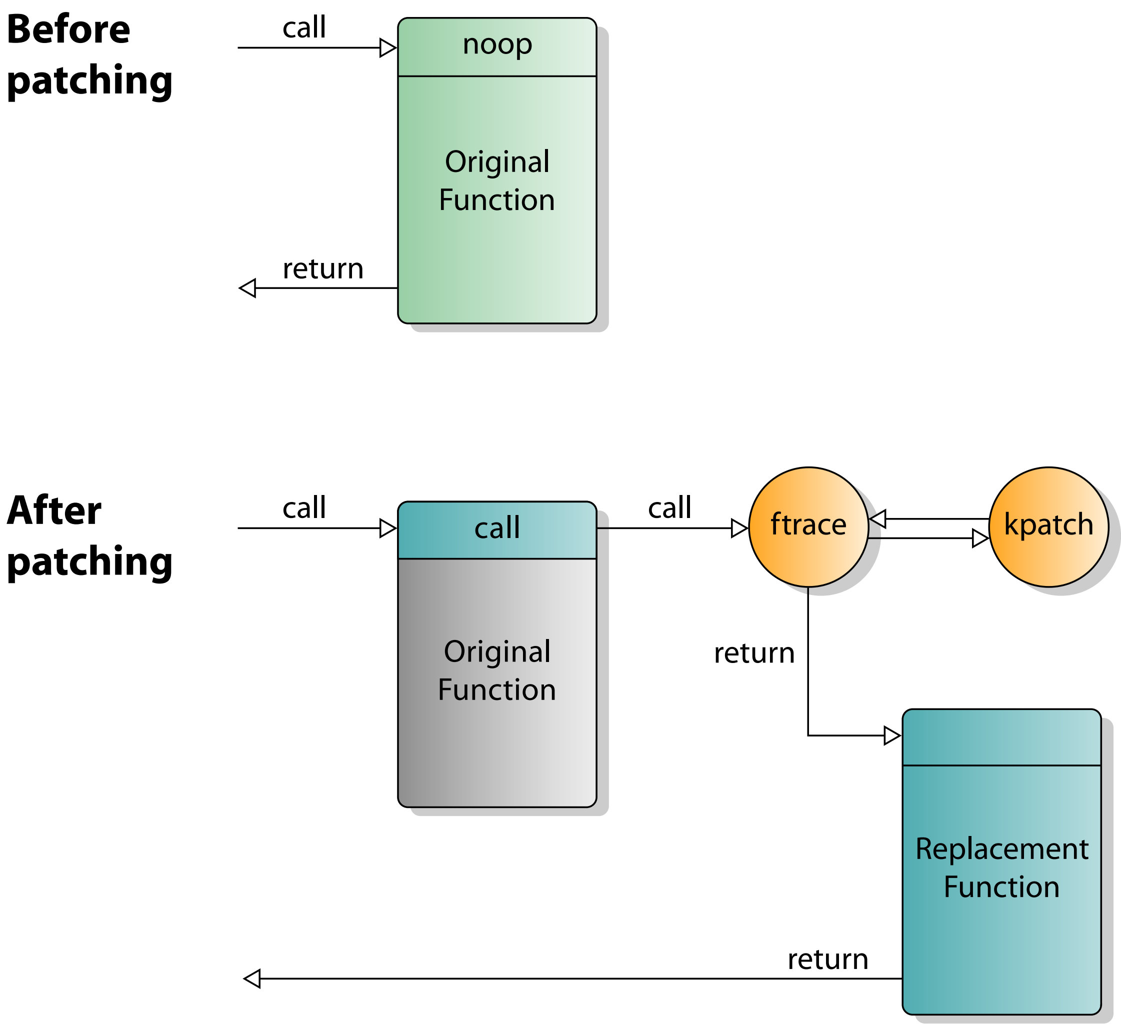
But what happens if a task is using the old function? Do we make a rule and say all tasks must be stopped, we patch, and then start them all again? Or do we add complexity by adding a list of tasks that use the old function, and tasks that use the new function, and maintain a trampoline which decides between each function for a given task?

What happens if the Livepatch changes the order that locks are acquired and released? The affected tasks which hold those locks need to be patched when the locks are no longer held, and the entire system needs to switch over to the new function at the same time. How do we co-ordinate this?

This is where consistency models come in, and is the driving force behind the different implementations of Livepatch. Each distribution has its own opinion on how things should be done, and we will look at all of them.

kpatch

kpatch is developed by Red Hat, and uses the simplest consistency model. kpatch operates pretty much as previously explained, by using ftrace to change the `nop` instruction in the old function to a `call` instruction, pointing to the new function.



kpatch keeps the system consistent by first stopping all running tasks. The stack traces of each task is then examined. If the old function is not found in any of the tasks stack traces, then ftrace applies the patch, and all future calls to the patched function will use the new function.

This approach is atomic and safe, since there is only one view of the function at a time, it is either old, or new. There are no consistency issues that arise if the new function changes data structures differently to the old function, and the structure is passed to tasks which haven't been migrated to the new function.

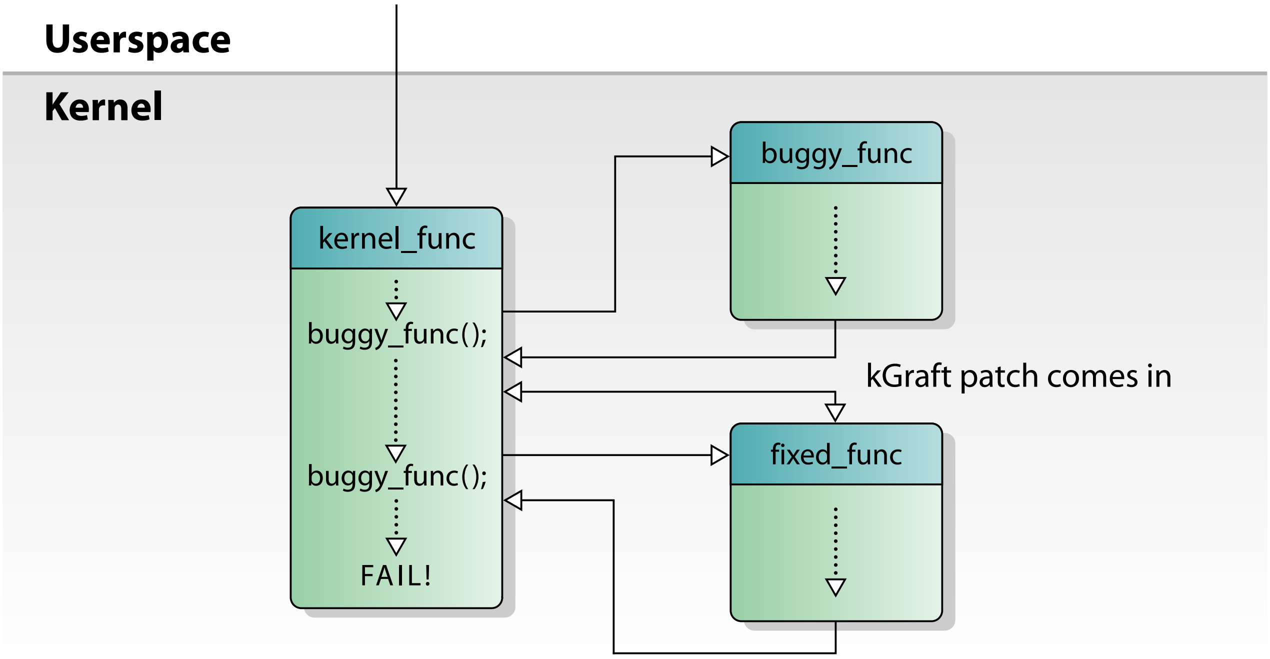
The limitations of kpatch involve not being able to modify data structures, and if a process is still using the patched function, patching fails, and all tasks are restarted again, to attempt the patch at a later time. There is some overhead in stopping and starting all tasks, which results in a small loss of service as those tasks are stopped.

kGraft

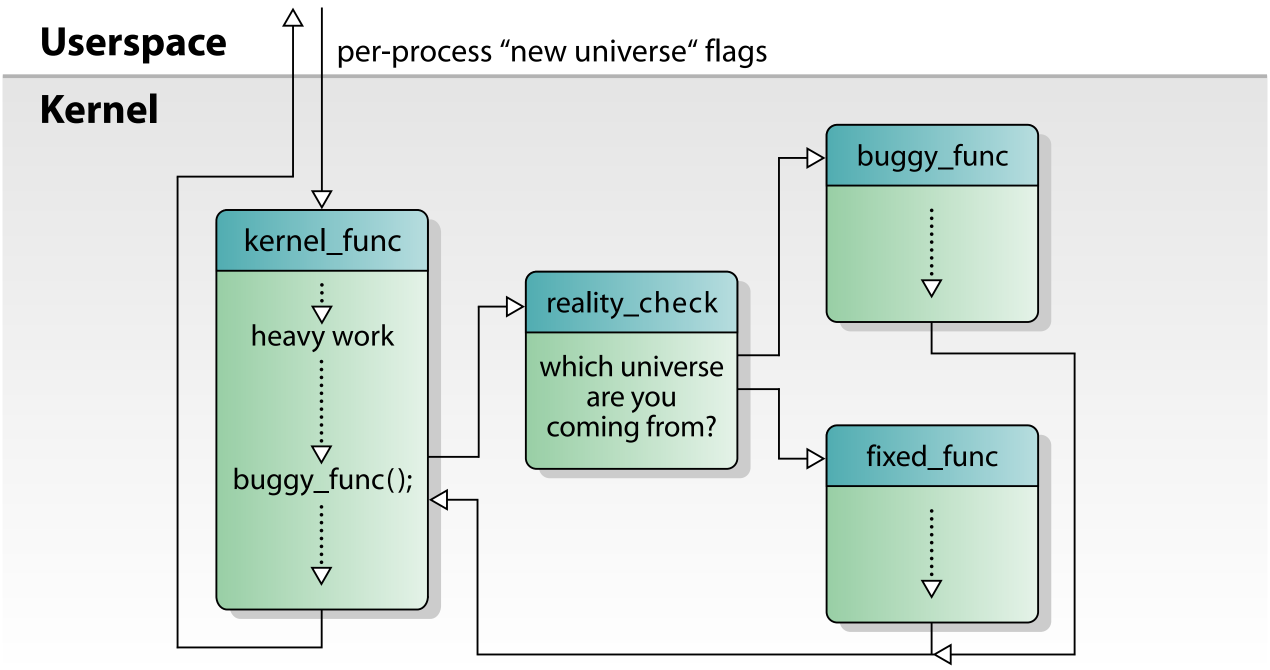
kGraft is developed by SUSE, and is by far the most complex consistency model. kGraft employs a per task consistency model, where all tasks remain running on the system, and tasks are patched one by one. This gives no downtime at all, since all tasks keep running during Livepatch, and patching can never "fail" in entirety.

kGraft achieves this by maintaining consistent "world views" to userspace processes, kernel threads and interrupt handlers, during their execution in kernel space.

For example, let's say we have a userspace process making a syscall, and a Livepatch request came in midway through this syscall.

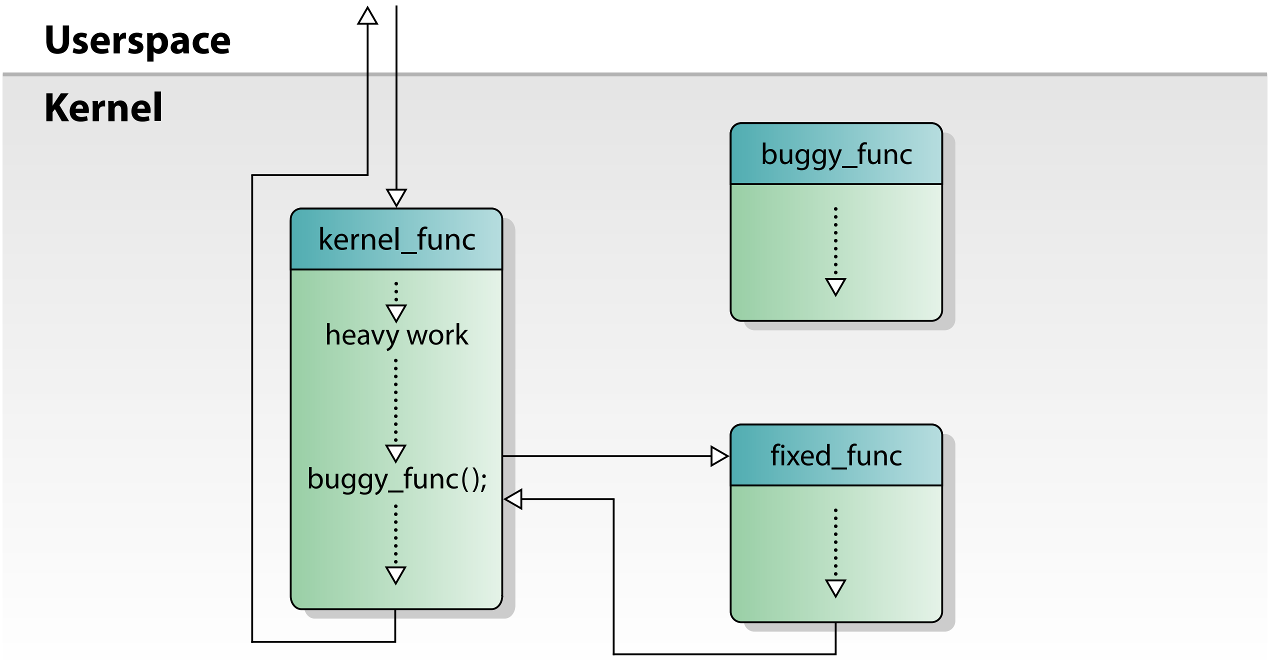


If the syscall involved calling the function which will be patched multiple times, on subsequent calling of the patched function, the semantics might have changed since the first time it was executed. If locking orders have changed, we might be facing a deadlock, which will end in certain failure.



Instead, what kGraft does is insert a trampoline which is the target of the `call` instruction which is replacing the `nop`. The trampoline points to both the old function and the new function. If the task has not yet been migrated to use the new function, the trampoline jumps to the old function and execution continues. If the task has been migrated, then the new function is called.

This means that any userspace process in a syscall, or kernel task, or interrupt handler still in kernel space will always use the old function.



This continues until each user space process finishes it syscall, or kernel task completes, or interrupt handler completes. At this stage, that task is then migrated over to the new function. When all tasks have been migrated, the trampoline is removed, and the `call` instruction is updated to point directly to the new function.

The benefits of kGraft is that all tasks are kept running during Livepatch. Downsides include keeping two different implementations of the same function around at the same time. This can cause problems when long running processes, like those waiting on disk or network I/O get stuck in kernel space, and won't be patched until they complete. This can lead to inconsistencies if the new function changes internal data structures differently to the original, since both functions can still be executed in parallel.

Ksplice

Ksplice is developed by Oracle, and has a consistency model similar to kpatch. Ksplice stops all tasks before patching the functions atomically.

The differentiating feature to Ksplice, is the ability to patch functions which require changes to data structures. This process is not automatic though, as a programmer must implement extra code to the Livepatch module which handles the transition from the old data structure to the new.

Livepatch (Mainline Linux)

Livepatch was mainlined into the Linux kernel during the 4.0 development cycle.

The Livepatch implementation is a hybrid between the kpatch and kGraft implementations, taking the best ideas from both. Livepatch uses kGraft's per task consistency and syscall exit migration, alongside kpatch's stack trace based switching.

Patches are applied on a per task basis, one task at a time. There is no downtime as tasks do not need to be stopped. This also means that the trampoline based solution is used.

The consistency model for mainline operates in a set of steps:

1. Firstly, the stack trace of sleeping tasks is checked. If the function to be patched is not found in the stack trace, the task is patched to use the new function. If this fails for a particular task, it will re-examine the stack trace periodically and attempt to patch at a later time. Most, if not all tasks will be patched in this step.
2. The second step is to patch the task once it completes and exits from kernel space, such as a syscall finishing or a interrupt handler completing. This is useful for long running I/O or cpubound tasks. In some cases, SIGSTOP must be issued to I/O bound tasks to force it to exit the kernel, be patched, and then send SIGCONT so it can continue.
3. For the kernel "swapper" task, which is executed whenever the CPU is idle and never exits the kernel, it has a special `klp_update_patch_state()` call in the idle loop which patches the task before the CPU enters the idle state.

What Consistency Model Does Ubuntu Use?

Ubuntu uses the Livepatch (mainline) consistency model, which has the best of both kpatch and kGraft. All code is the same as what is shipped in the mainline kernel, and there are no custom changes.

Writing our Own Livepatches

Now that we have learned a bit about what Livepatch is, how it works, and the careful consideration that goes into selecting a consistency model, let's start making some Livepatches of our own.

Structure of a Livepatch

For our first Livepatch, I think we will follow the sample which is provided in the mainline kernel. Download a copy of livepatch-sample.c and have a read.

Note, the Livepatch API has changed over time, so if you want to build for 4.4 Xenial, use the `livepatch-sample.c` from the Xenial kernel sources. If you get an error `insmod: ERROR:`

`could not insert module livepatch-sample.ko: Invalid parameters` then you are using the wrong Livepatch API.

I am going to explain the latest API, as found in 5.4 Focal.

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/livepatch.h>

#include <linux/seq_file.h>
static int livepatch_cmdline_proc_show(struct seq_file *m, void *v)
{
        seq_printf(m, "%s\n", "this has been live patched");
        return 0;
}

static struct klp_func funcs[] = {
        {
                .old_name = "cmdline_proc_show",
                .new_func = livepatch_cmdline_proc_show,
        }, { }
};

static struct klp_object objs[] = {
        {
                /* name being NULL means vmlinux */
                .funcs = funcs,
        }, { }
};

static struct klp_patch patch = {
        .mod = THIS_MODULE,
        .objs = objs,
};

static int livepatch_init(void)
{
        return klp_enable_patch(&patch);
}

static void livepatch_exit(void)
{
}

module_init(livepatch_init);
module_exit(livepatch_exit);
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y");
```

As you can already see, since the Livepatch is a kernel module, it follows the same process required when writing a kernel module. We `#include` the kernel module header files of `linux/module.h` and `linux/kernel.h`, and declare our `module_init()` and `module_exit()` function pointers.

To say we are making a Livepatch, we also include `linux/livepatch.h`, set the module info marco to `livepatch, Y` and have the module init function call `klp_enable_patch()`, the entry point to the Livepatch subsystem.

Declaring the Livepatch itself is pretty simple. In this example, we will patch `cmdline_proc_show()`, the function which retruns the kernel commandline when you read from `/proc/cmdline`.

We define a new function, `livepatch_cmdline_proc_show()`, and give the "fixed" implementation. We then map the new function to the old function by defining a struct of type `klp_func`, in this case called `funcs[]`, and filling in the members `.old_name` and `.new_func`.

Since we might need to replace more than one function in our Livepatch, we can create many of these function mappings, since `funcs[]` is an array.

We then tell Livepatch what to patch with struct `klp_object`. We set `.funcs` to our array of functions, and set `.name` to be another Livepatch module this has a dependency on, or simply `NULL` if we want to target `vmlinux`.

Finally, this is wrapped into a struct `klp_patch`, where we declare the module name, and the object struct. This is the struct we pass a reference to when `klp_enable_patch()` is called.

We can build the module with the following `Makefile`:

```
obj-m := livepatch-sample.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

You need to install a compiler, and the kernel header for your running kernel:

```
$ sudo apt install linux-headers-`uname -r`
$ sudo apt install build-essential
```

Then go ahead and run `make`:

```
$ make
make -C /lib/modules/5.4.0-21-generic/build M=/home/ubuntu/simple modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-21-generic'
```

```
  CC [M]  /home/ubuntu/simple/livepatch-sample.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/ubuntu/simple/livepatch-sample.mod.o
  LD [M]  /home/ubuntu/simple/livepatch-sample.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-21-generic'
```

I did this on Focal, but this should work on any Ubuntu kernel from 4.4 Xenial and upward, as they all have Livepatch enabled.

We then have the end result, `livepatch-sample.ko`. Lets do a before and after read of `/proc/cmdline` as we load the module:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.4.0-21-generic root=UUID=f9f909c3-782a-43c2-a59d-c78
$ sudo insmod livepatch-sample.ko
$ cat /proc/cmdline
this has been live patched
```

How cool is that? We have successfully Livepatched our system. Checking `dmesg` shows us the progress of Livepatch:

```
[   33.100762] livepatch_sample: loading out-of-tree module taints kernel.
[   33.100764] livepatch_sample: tainting kernel with TAINT_LIVEPATCH
[   33.100793] livepatch_sample: module verification failed: signature and/or r
[   33.111720] livepatch: enabling patch 'livepatch_sample'
[   33.114679] livepatch: 'livepatch_sample': starting patching transition
[   33.883586] livepatch: 'livepatch_sample': patching complete
```

Note, we didn’t sign our kernel module, which is why module verification failed. This is only really important if you are using secureboot. Otherwise, our kernel gained taint flags for loading the Livepatch module.

Making a Slightly More Complex Livepatch

The previous Livepatch example used a completely new basic function to write back a replaced kernel command line. What happens if we want to actually patch existing code?

The next example will follow along the case for using kpatch-build, using the primary example in the kpatch repository.

What we want to do is change how the text is displayed for `VmallocChunk` in `/proc/meminfo`. The following patch for Linux 5.4 makes it capitalised:

```
diff --git a/fs/proc/meminfo.c b/fs/proc/meminfo.c
index 8c1f1bb1a5ce..3053c1bce50d 100644
--- a/fs/proc/meminfo.c
+++ b/fs/proc/meminfo.c
@@ -117,7 +117,7 @@ static int meminfo_proc_show(struct seq_file *m, void *v)
        seq_printf(m, "VmallocTotal:   %8lu kB\n",
                   (unsigned long)VMALLOC_TOTAL >> 10);
        show_val_kb(m, "VmallocUsed:    ", vmalloc_nr_pages());
-       show_val_kb(m, "VmallocChunk:   ", 0ul);
+       show_val_kb(m, "VMALLOCCHUNK:   ", 0ul);
        show_val_kb(m, "Percpu:         ", pcpu_nr_pages());

 #ifdef CONFIG_MEMORY_FAILURE
```

Writing the Livepatch Ourselves

Okay, let's follow a similar format to last time. Let's copy the new function into our Livepatch template, like so:

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/livepatch.h>

static int livepatch_meminfo_proc_show(struct seq_file *m, void *v)
{
        struct sysinfo i;
        unsigned long committed;
        long cached;
        long available;
        unsigned long pages[NR_LRU_LISTS];
        unsigned long sreclaimable, sunreclaim;
        int lru;

        si_meminfo(&i);
        si_swapinfo(&i);
        committed = percpu_counter_read_positive(&vm_committed_as);

        cached = global_node_page_state(NR_FILE_PAGES) -
                        total_swapcache_pages() - i.bufferram;
        if (cached < 0)
                cached = 0;

        for (lru = LRU_BASE; lru < NR_LRU_LISTS; lru++)
                pages[lru] = global_node_page_state(NR_LRU_BASE + lru);

        available = si_mem_available();
        sreclaimable = global_node_page_state(NR_SLAB_RECLAIMABLE);
        sunreclaim = global_node_page_state(NR_SLAB_UNRECLAIMABLE);

        show_val_kb(m, "MemTotal:       ", i.totalram);
        show_val_kb(m, "MemFree:        ", i.freeram);
        show_val_kb(m, "MemAvailable:   ", available);
        show_val_kb(m, "Buffers:        ", i.bufferram);
```

```
        show_val_kb(m, "Cached:         ", cached);
        show_val_kb(m, "SwapCached:     ", total_swapcache_pages());
        show_val_kb(m, "Active:         ", pages[LRU_ACTIVE_ANON] +
                                           pages[LRU_ACTIVE_FILE]);
        show_val_kb(m, "Inactive:       ", pages[LRU_INACTIVE_ANON] +
                                           pages[LRU_INACTIVE_FILE]);
        show_val_kb(m, "Active(anon):   ", pages[LRU_ACTIVE_ANON]);
        show_val_kb(m, "Inactive(anon): ", pages[LRU_INACTIVE_ANON]);
        show_val_kb(m, "Active(file):   ", pages[LRU_ACTIVE_FILE]);
        show_val_kb(m, "Inactive(file): ", pages[LRU_INACTIVE_FILE]);
        show_val_kb(m, "Unevictable:    ", pages[LRU_UNEVICTABLE]);
        show_val_kb(m, "Mlocked:        ", global_zone_page_state(NR_MLOCK));

#ifdef CONFIG_HIGHMEM
        show_val_kb(m, "HighTotal:      ", i.totalhigh);
        show_val_kb(m, "HighFree:       ", i.freehigh);
        show_val_kb(m, "LowTotal:       ", i.totalram - i.totalhigh);
        show_val_kb(m, "LowFree:        ", i.freeram - i.freehigh);
#endif

#ifndef CONFIG_MMU
        show_val_kb(m, "MmapCopy:       ",
                    (unsigned long)atomic_long_read(&mmap_pages_allocated));
#endif

        show_val_kb(m, "SwapTotal:      ", i.totalswap);
        show_val_kb(m, "SwapFree:       ", i.freeswap);
        show_val_kb(m, "Dirty:          ",
                    global_node_page_state(NR_FILE_DIRTY));
        show_val_kb(m, "Writeback:      ",
                    global_node_page_state(NR_WRITEBACK));
        show_val_kb(m, "AnonPages:      ",
                    global_node_page_state(NR_ANON_MAPPED));
        show_val_kb(m, "Mapped:         ",
                    global_node_page_state(NR_FILE_MAPPED));
        show_val_kb(m, "Shmem:          ", i.sharedram);
        show_val_kb(m, "KReclaimable:   ", sreclaimable +
                    global_node_page_state(NR_KERNEL_MISC_RECLAIMABLE));
        show_val_kb(m, "Slab:           ", sreclaimable + sunreclaim);
        show_val_kb(m, "SReclaimable:   ", sreclaimable);
        show_val_kb(m, "SUnreclaim:     ", sunreclaim);
        seq_printf(m, "KernelStack:    %8lu kB\n",
                   global_zone_page_state(NR_KERNEL_STACK_KB));
        show_val_kb(m, "PageTables:     ",
                    global_zone_page_state(NR_PAGETABLE));

        show_val_kb(m, "NFS_Unstable:   ",
                    global_node_page_state(NR_UNSTABLE_NFS));
        show_val_kb(m, "Bounce:         ",
                    global_zone_page_state(NR_BOUNCE));
        show_val_kb(m, "WritebackTmp:   ",
                    global_node_page_state(NR_WRITEBACK_TEMP));
        show_val_kb(m, "CommitLimit:    ", vm_commit_limit());
        show_val_kb(m, "Committed_AS:   ", committed);
        seq_printf(m, "VmallocTotal:   %8lu kB\n",
                   (unsigned long)VMALLOC_TOTAL >> 10);
        show_val_kb(m, "VmallocUsed:    ", vmalloc_nr_pages());
        show_val_kb(m, "VMALLOCCHUNK:   ", 0ul);
        show_val_kb(m, "Percpu:         ", pcpu_nr_pages());

#ifdef CONFIG_MEMORY_FAILURE
```

```
	seq_printf(m, "HardwareCorrupted: %5lu kB\n",
		   atomic_long_read(&num_poisoned_pages) << (PAGE_SHIFT - 10));
#endif

#ifdef CONFIG_TRANSPARENT_HUGEPAGE
	show_val_kb(m, "AnonHugePages:  ",
		    global_node_page_state(NR_ANON_THPS) * HPAGE_PMD_NR);
	show_val_kb(m, "ShmemHugePages: ",
		    global_node_page_state(NR_SHMEM_THPS) * HPAGE_PMD_NR);
	show_val_kb(m, "ShmemPmdMapped: ",
		    global_node_page_state(NR_SHMEM_PMDMAPPED) * HPAGE_PMD_NR);
	show_val_kb(m, "FileHugePages:  ",
		    global_node_page_state(NR_FILE_THPS) * HPAGE_PMD_NR);
	show_val_kb(m, "FilePmdMapped:  ",
		    global_node_page_state(NR_FILE_PMDMAPPED) * HPAGE_PMD_NR);
#endif

#ifdef CONFIG_CMA
	show_val_kb(m, "CmaTotal:       ", totalcma_pages);
	show_val_kb(m, "CmaFree:        ",
		    global_zone_page_state(NR_FREE_CMA_PAGES));
#endif

	hugetlb_report_meminfo(m);

	arch_report_meminfo(m);

	return 0;
}

static struct klp_func funcs[] = {
	{
		.old_name = "meminfo_proc_show",
		.new_func = livepatch_meminfo_proc_show,
	}, { }
};

static struct klp_object objs[] = {
	{
		/* name being NULL means vmlinux */
		.funcs = funcs,
	}, { }
};

static struct klp_patch patch = {
	.mod = THIS_MODULE,
	.objs = objs,
};

static int livepatch_init(void)
{
	return klp_enable_patch(&patch);
}

static void livepatch_exit(void)
{
}

module_init(livepatch_init);
module_exit(livepatch_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y");
```

We can pretty much keep the same `Makefile` as last time:

```
obj-m := livepatch-meminfo.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

When we build, we see some unresolved symbols:

```
$ make
make -C /lib/modules/5.4.0-21-generic/build M=/home/ubuntu/meminfo modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-21-generic'
  CC [M]  /home/ubuntu/meminfo/livepatch-meminfo.o
/home/ubuntu/meminfo/livepatch-meminfo.c: In function ‘livepatch_meminfo_proc_s
/home/ubuntu/meminfo/livepatch-meminfo.c:19:9: error: implicit declaration of f
   19 |         si_swapinfo(&i);
      |         ^~~~~~~~~~~
/home/ubuntu/meminfo/livepatch-meminfo.c:20:51: error: ‘vm_committed_as’ undecl
   20 |         committed = percpu_counter_read_positive(&vm_committed_as);
      |                                                   ^~~~~~~~~~~~~~~
/home/ubuntu/meminfo/livepatch-meminfo.c:20:51: note: each undeclared identifie
/home/ubuntu/meminfo/livepatch-meminfo.c:23:25: error: implicit declaration of
   23 |                         total_swapcache_pages() - i.bufferram;
      |                         ^~~~~~~~~~~~~~~~~~~~~
/home/ubuntu/meminfo/livepatch-meminfo.c:34:9: error: implicit declaration of f
   34 |         show_val_kb(m, "MemTotal:       ", i.totalram);
      |         ^~~~~~~~~~~
/home/ubuntu/meminfo/livepatch-meminfo.c:90:44: error: implicit declaration of
   90 |         show_val_kb(m, "CommitLimit:    ", vm_commit_limit());
      |                                            ^~~~~~~~~~~~~~~
/home/ubuntu/meminfo/livepatch-meminfo.c:117:44: error: ‘totalcma_pages’ undecl
  117 |         show_val_kb(m, "CmaTotal:       ", totalcma_pages);
      |                                            ^~~~~~~~~~~~~~
      |                                            totalram_pages
/home/ubuntu/meminfo/livepatch-meminfo.c:122:9: error: implicit declaration of
  122 |         hugetlb_report_meminfo(m);
      |         ^~~~~~~~~~~~~~~~~~~~~~
      |         arch_report_meminfo
cc1: some warnings being treated as errors
make[2]: *** [scripts/Makefile.build:275: /home/ubuntu/meminfo/livepatch-meminf
make[1]: *** [Makefile:1719: /home/ubuntu/meminfo] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-21-generic'
make: *** [Makefile:5: default] Error 2
```

Not to worry! We are just missing some header files. Look at the symbols and use cscope to find what header files they live in, and `#include` them:

```
#include <linux/seq_file.h>
#include <linux/swap.h>
#include <linux/mman.h>
#include <linux/cma.h>
#include <linux/hugetlb.h>
```

Now lets build:

```
$ make
make -C /lib/modules/5.4.0-21-generic/build M=/home/ubuntu/meminfo modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-21-generic'
  CC [M]  /home/ubuntu/meminfo/livepatch-meminfo.o
/home/ubuntu/meminfo/livepatch-meminfo.c: In function ‘livepatch_meminfo_proc_s
/home/ubuntu/meminfo/livepatch-meminfo.c:38:9: error: implicit declaration of f
   38 |         show_val_kb(m, "MemTotal:       ", i.totalram);
      |         ^~~~~~~~~~~
cc1: some warnings being treated as errors
make[2]: *** [scripts/Makefile.build:275: /home/ubuntu/meminfo/livepatch-meminf
make[1]: *** [Makefile:1719: /home/ubuntu/meminfo] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-21-generic'
make: *** [Makefile:5: default] Error 2
```

Unfortunately for us, this basic example calls `show_val_kb()`. This isn't defined in any header files, and is actually local to `fs/proc/meminfo.c`.

```
static void show_val_kb(struct seq_file *m, const char *s, unsigned long num)
{
    seq_put_decimal_ull_width(m, s, num << (PAGE_SHIFT - 10), 8);
    seq_write(m, " kB\n", 4);
}
```

So close but so far! Now, these functions which are local to their modules don't actually export their symbols to a stripped vmlinuz, which means we have a problem. Even if we try be cheeky and make a forward declaration and label it `extern`:

```
extern void show_val_kb(struct seq_file *m, const char *s, unsigned long num);
```

The compiler is onto us!

```
$ make
make -C /lib/modules/5.4.0-21-generic/build M=/home/ubuntu/meminfo modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-21-generic'
  CC [M]  /home/ubuntu/meminfo/livepatch-meminfo.o
  Building modules, stage 2.
  MODPOST 1 modules
ERROR: "arch_report_meminfo" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefi
ERROR: "hugetlb_report_meminfo" [/home/ubuntu/meminfo/livepatch-meminfo.ko] und
ERROR: "totalcma_pages" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
ERROR: "num_poisoned_pages" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefin
ERROR: "pcpu_nr_pages" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
ERROR: "vmalloc_nr_pages" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined
ERROR: "vm_commit_limit" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
ERROR: "show_val_kb" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
ERROR: "total_swapcache_pages" [/home/ubuntu/meminfo/livepatch-meminfo.ko] unde
ERROR: "vm_committed_as" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
ERROR: "si_swapinfo" [/home/ubuntu/meminfo/livepatch-meminfo.ko] undefined!
make[2]: *** [scripts/Makefile.modpost:94: __modpost] Error 1
make[1]: *** [Makefile:1632: modules] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-21-generic'
make: *** [Makefile:5: default] Error 2
```

While the module object builds, it cannot be linked, since the compiler does not know the offsets or locations of the functions which reside in the unstripped vmlinux / stripped vmlinuz binaries.

So, how do we fix this? I struggled with this issue for quite a long time, until I went back and read the Livepatch documentation more closely.

From Documentation/livepatch/livepatch.txt:

```
The patch contains only functions that are really modified. But they
might want to access functions or data from the original source file
that may only be locally accessible. This can be solved by a special
relocation section in the generated livepatch module, see
Documentation/livepatch/module-elf-format.txt for more details.
```

If you go ahead and read Documentation/livepatch/module-elf-format.txt, we find that we need to add ELF sections to the object file which tell the kernel Livepatch subsystem how to apply relocations for each of these functions into the kernel we are targeting.

There are two ELF sections that need adding;

- SHF_RELA_LIVEPATCH
- SHN_LIVEPATCH

SHF_RELA_LIVEPATCH is used to declare the functions which need to be redirected with ftrace, that is, the functions that are actually being Livepatched.

SHN_LIVEPATCH are all the local symbols that the fixed function calls, and need to be fixed up.

Each section needs entries of the from:

```
.klp.rela.objname.section_name
```

An example for `SHF_RELA_LIVEPATCH` would be:

```
.klp.rela.vmlinux.text.meminfo.proc_show
```

These ELF sections need to know the addresses and offsets from the vmlinux binary.

Now, inserting these by hand is actually really hard, and does not scale at all.

This is the idea behind `kpatch-build` , and automated build program which can generate Livepatches from source diffs, and programatically fetch and insert these ELF sections which contain the symbol relocation tables.

Using kpatch-build to Generate the Livepatch

Firstly we need to download and build kpatch-build:

```
$ sudo apt install dpkg-dev devscripts elfutils ccache
$ sudo apt build-dep linux
$ git clone https://github.com/dynup/kpatch.git
$ cd kpatch
$ make
```

The next step is to download the `ddeb` (debug-deb) package for the kernel we wish to make a Livepatch module for. A list of all kernel ddeb packages can be found at the ddeb package repository.

I will be targeting 5.4.0-24-generic, so I need to download linux-image-unsigned-5.4.0-24-generic-dbgsym_5.4.0-24.28_amd64.ddeb.

```
$ wget http://ddebs.ubuntu.com/ubuntu/pool/main/l/linux/linux-image-unsigned-5.
$ sudo dpkg -i linux-image-unsigned-5.4.0-24-generic-dbgsym_5.4.0-24.28_amd64.d
```

The resulting debug vmlinux will be placed at `/lib/debug/boot/vmlinux-5.4.0-24-generic`.

`kpatch-build` operates on source diffs. Save the diff to `~/meminfo-string.patch` like so:

```
$ cat ~/meminfo-string.patch
diff --git a/fs/proc/meminfo.c b/fs/proc/meminfo.c
index 8c1f1bb1a5ce..3053c1bce50d 100644
--- a/fs/proc/meminfo.c
+++ b/fs/proc/meminfo.c
@@ -117,7 +117,7 @@ static int meminfo_proc_show(struct seq_file *m, void *v)
        seq_printf(m, "VmallocTotal:   %8lu kB\n",
                   (unsigned long)VMALLOC_TOTAL >> 10);
        show_val_kb(m, "VmallocUsed:    ", vmalloc_nr_pages());
-       show_val_kb(m, "VmallocChunk:   ", 0ul);
+       show_val_kb(m, "VMALLOCCHUNK:   ", 0ul);
        show_val_kb(m, "Percpu:         ", pcpu_nr_pages());

 #ifdef CONFIG_MEMORY_FAILURE
```

Now we are ready to build!

Run the following command:

```
$ kpatch/kpatch-build/kpatch-build -t vmlinux --vmlinux /lib/debug/boot/vmlinux
Using cache at /home/matthew/.kpatch/src
Testing patch file(s)
Reading special section data
readelf: Error: LEB value too large
readelf: Error: LEB value too large
Building original source
Building patched source
Extracting new and modified ELF sections
meminfo.o: changed function: meminfo_proc_show
Patched objects: vmlinux
Building patch module: livepatch-meminfo-string.ko
SUCCESS
```

`kpatch-build` works by first downloading the source archive of the kernel you are targeting, which is determined by the vmlinux package you pass in. From there, the standard vmlinux is built normally. Once that completes, the source tree is patched with the patch you specified, and rebuilt. Since most patches are small, only changed object files are rebuilt. In this case, only `meminfo.o` gets rebuilt.

Since we now know that only `meminfo.o` got changed, the single object is compiled again with `-ffunction-sections -fdata-sections` in both the patched and unpatched forms.

Then each unpatched and patched object set is then analysed by `create-diff-object` to determine what functions have been modified, and to extract the changed functions. This program also checks for Livepatch compatibility.

The really special part of `create-diff-object` is that it creates the necessary ELF symbol relocation sections to the patched objectfile.

It adds `kpatch.funcs` and `.rela.kpatch.funcs` which tell ftrace what functions are actually going to be Livepatched.

It adds `.kpatch.dynrelas` and `.rela.kpatch.dynrelas` which are used to fixup symbol relocations for local function calls in the fixed function to symbols in vmlinux.

From there, `kpatch-build` generates a new kernel module containing all Livepatches, which is ready to be used.

Let's test it out shall we?

```
$ sudo insmod livepatch-meminfo-string.ko
$ grep -i chunk /proc/meminfo
VMALLOCCHUNK:          0 kB
```

It worked! Great! Let's see what `dmesg` has to say:

```
[ 5611.674220] livepatch_meminfo_string: loading out-of-tree module taints kern
[ 5611.674223] livepatch_meminfo_string: tainting kernel with TAINT_LIVEPATCH
[ 5611.674259] livepatch_meminfo_string: module verification failed: signature
[ 5611.856109] livepatch: enabling patch 'livepatch_meminfo_string'
[ 5611.859603] livepatch: 'livepatch_meminfo_string': starting patching transit
[ 5611.860277] livepatch: 'livepatch_meminfo_string': patching complete
```

Pretty much the same as last time.

As for those ELF sections, we can examine the kernel module to see them:

```
$ readelf --sections livepatch-meminfo-string.ko
There are 52 section headers, starting at offset 0xac7e8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [20] .kpatch.funcs     PROGBITS         0000000000000000  00001fa8
       0000000000000038  0000000000000000   A       0     0     8
  [21] .rela.kpatch.func RELA             0000000000000000  00001fe0
       0000000000000048  0000000000000018   I      48    20     8
```

```
...
  [51] .klp.rela.vmlinux RELA             0000000000000000  000ac308
       00000000000004e0  0000000000000018 AIo      48    10     8
```

```
$ readelf --relocs livepatch-meminfo-string.ko
...
Relocation section '.klp.rela.vmlinux..text.meminfo_proc_show' at offset 0xac30
  Offset          Info           Type           Sym. Value    Sym. Name + Adden
00000000003f  005400000004 R_X86_64_PLT32    0000000000000000 .klp.sym.vmlinux.
000000000046  005500000002 R_X86_64_PC32     0000000000000000 .klp.sym.vmlinux.
...
```

Using Livepatch to Fix A Real Bug

Now, I really wanted to make a Livepatch to fix a real bug, but for the moment I must admit defeat.

I went into writing this blog post thinking that Livepatch could be an awesome tool to help fix customer issues, but the problem is, there are some severe limitations as to what can be Livepatched, and even when you believe a patch could be compatible, a GCC optimisation could completely ruin your plans.

I have two examples.

Example One: Inline Functions

The first, is a bug that was actually a regression to the SRU I made for the bug fixed by my previous blog post, Resolving Large NVMe Performance Degradation in the Ubuntu 4.4 Kernel.

Anyway, the bug is documented by my colleague who I worked the case with:

Mounting LVM snapshots with xfs can hit kernel BUG in nvme driver.

```
commit 5a8d75a1b8c99bdc926ba69b7b7dbe4fae81a5af
Author: Ming Lei <ming.lei@redhat.com>
Date:   Fri Apr 14 13:58:29 2017 -0600
Subject: block: fix bio_will_gap() for first bvec with offset
```

You can read the commit here: block: fix bio_will_gap() for first bvec with offset.

The important part is the three function prototypes in each changed function:

```
-static inline bool bio_will_gap(struct request_queue *q, struct bio *prev,
-                                struct bio *next)
+static inline bool bio_will_gap(struct request_queue *q,
+                                struct request *prev_rq,
+                                struct bio *prev,
+                                struct bio *next)

static inline bool req_gap_back_merge(struct request *req, struct bio *bio)

static inline bool req_gap_front_merge(struct request *req, struct bio *bio)
```

Inlined functions. Sometimes these will work, as the callers will just embed the code in them. Most of the time they won't though.

The thing is, the kernel redefines the meaning of `inline` in `include/linux/compiler_types.h`:

```
#if !defined(CONFIG_OPTIMIZE_INLINING)
#define inline inline __attribute__((__always_inline__)) __gnu_inline \
        __inline_maybe_unused notrace
#else
#define inline inline                                    __gnu_inline \
        __inline_maybe_unused notrace
#endif
```

We see that if you select `inline`, you also get `notrace`. Only tracable functions can be Livepatched as we know, meaning that this is a dead end if you are not using tools like `kpatch-build`. Most patches like this will mostly error out with `kpatch-build` too.

Example Two: GCC Optimisations

The next bug is a neat little Null pointer dereference if you have the sysctl `kernel.core_pattern` set to "|" and run a program which crashes.

You can read all about it here:

unkillable process (kernel NULL pointer dereference)

There's a patch made by Sudip Mukherjee which was more elegant than the one I put forward in the process of getting mainlined now. You can see it here:

```
diff --git a/fs/coredump.c b/fs/coredump.c
index f8296a82d01d..408418e6aa13 100644
--- a/fs/coredump.c
+++ b/fs/coredump.c
@@ -211,6 +211,8 @@ static int format_corename(struct core_name *cn, struct cor
                        return -ENOMEM;
```

```
                (*argv)[(*argc)++] = 0;
                ++pat_ptr;
+               if (!(*pat_ptr))
+                       return -ENOMEM;
        }

        /* Repeat as long as we have more pattern to process and more output
```

Now, if we run `kpatch-build` over this:

```
$ kpatch/kpatch-build/kpatch-build -t vmlinux --vmlinux /lib/debug/boot/vmlinux
Using cache at /home/matthew/.kpatch/src
Testing patch file(s)
Reading special section data
readelf: Error: LEB value too large
readelf: Error: LEB value too large
Building original source
Building patched source
Extracting new and modified ELF sections
coredump.o: changed function: do_coredump
/home/matthew/work/kernel/kpatch/kpatch-build/create-diff-object: ERROR: coredu
ERROR: 1 error(s) encountered. Check /home/matthew/.kpatch/build.log for more d
```

It fails! Why does it say the changed function was `do_coredump()` , when the above patch clearly patches `format_corename()` ? There are no inlined functions here.

To get some answers, we need to look at the vmlinux binaries to see what symbols are exported.

```
$ readelf -s /lib/debug/boot/vmlinux-5.4.0-24-generic
...
 29993: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS coredump.c
 29994: ffffffff8247f938     0 NOTYPE  LOCAL  DEFAULT   13 __ksymtab_dump_emit
 29995: ffffffff824a80eb    10 OBJECT  LOCAL  DEFAULT   17 __kstrtab_dump_emit
 29996: ffffffff8247f95c     0 NOTYPE  LOCAL  DEFAULT   13 __ksymtab_dump_skip
 29997: ffffffff824a80e1    10 OBJECT  LOCAL  DEFAULT   17 __kstrtab_dump_skip
 29998: ffffffff8247f92c     0 NOTYPE  LOCAL  DEFAULT   13 __ksymtab_dump_align
 29999: ffffffff824a80d6    11 OBJECT  LOCAL  DEFAULT   17 __kstrtab_dump_align
 30000: ffffffff8247f974     0 NOTYPE  LOCAL  DEFAULT   13 __ksymtab_dump_trunc
 30001: ffffffff824a80c8    14 OBJECT  LOCAL  DEFAULT   17 __kstrtab_dump_trunc
 30002: ffffffff813610b0   156 FUNC    LOCAL  DEFAULT    1 umh_pipe_setup
 30003: ffffffff81361150   208 FUNC    LOCAL  DEFAULT    1 zap_process
 30004: ffffffff813612e0   100 FUNC    LOCAL  DEFAULT    1 expand_corename.isra
 30005: ffffffff827144c0     4 OBJECT  LOCAL  DEFAULT   24 core_name_size
 30006: ffffffff81361350   195 FUNC    LOCAL  DEFAULT    1 cn_vprintf
 30007: ffffffff81361420   106 FUNC    LOCAL  DEFAULT    1 cn_printf
 30008: ffffffff81361490   247 FUNC    LOCAL  DEFAULT    1 cn_esc_printf
 30009: ffffffff82d3f560  4096 OBJECT  LOCAL  DEFAULT   54 zeroes.62762
 30010: ffffffff81361660  1383 FUNC    LOCAL  DEFAULT    1 format_corename.isra
 30011: ffffffff81361bd0    36 FUNC    LOCAL  DEFAULT    1 kmalloc_array.constp
 30012: ffffffff82d40560     0 OBJECT  LOCAL  DEFAULT   54 __key.10435
 30013: ffffffff82d40560     4 OBJECT  LOCAL  DEFAULT   54 core_dump_count.6271
```

```
 30014: ffffffff81362730    56 FUNC    LOCAL  DEFAULT    1 do_coredump.cold
 30015: ffffffff82079530    12 OBJECT  LOCAL  DEFAULT    7 __func__.62732
...
```

Next, the freshly built vmlinux:

```
$ readelf -s ~/.kpatch/src/vmlinux
...
 92711: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS coredump.c
 92712: ffffffff8248f918     0 NOTYPE  LOCAL  DEFAULT 97899 __ksymtab_dump_emit
 92713: ffffffff824b80cb    10 OBJECT  LOCAL  DEFAULT 97903 __kstrtab_dump_emit
 92714: ffffffff8248f93c     0 NOTYPE  LOCAL  DEFAULT 97899 __ksymtab_dump_skip
 92715: ffffffff824b80c1    10 OBJECT  LOCAL  DEFAULT 97903 __kstrtab_dump_skip
 92716: ffffffff8248f90c     0 NOTYPE  LOCAL  DEFAULT 97899 __ksymtab_dump_alig
 92717: ffffffff824b80b6    11 OBJECT  LOCAL  DEFAULT 97903 __kstrtab_dump_alig
 92718: ffffffff8248f954     0 NOTYPE  LOCAL  DEFAULT 97899 __ksymtab_dump_trun
 92719: ffffffff824b80a8    14 OBJECT  LOCAL  DEFAULT 97903 __kstrtab_dump_trun
 92720: ffffffff814baff0   156 FUNC    LOCAL  DEFAULT 8647 umh_pipe_setup
 92721: ffffffff81761a10   208 FUNC    LOCAL  DEFAULT 32162 zap_process
 92722: ffffffff81761ba0   100 FUNC    LOCAL  DEFAULT 32166 expand_corename.isr
 92723: ffffffff8276d518     4 OBJECT  LOCAL  DEFAULT 106303 core_name_size
 92724: ffffffff81761c10   195 FUNC    LOCAL  DEFAULT 32168 cn_vprintf
 92725: ffffffff81761ce0   106 FUNC    LOCAL  DEFAULT 32170 cn_printf
 92726: ffffffff81761d50   247 FUNC    LOCAL  DEFAULT 32172 cn_esc_printf
 92727: ffffffff83017f60  4096 OBJECT  LOCAL  DEFAULT 117495 zeroes.62762
 92728: ffffffff82ec62d0     0 OBJECT  LOCAL  DEFAULT 116793 __key.10435
 92729: ffffffff83018f60     4 OBJECT  LOCAL  DEFAULT 117496 core_dump_count.62
 92730: ffffffff81761f16    27 FUNC    LOCAL  DEFAULT 32178 do_coredump.cold
 92731: ffffffff822adba0    12 OBJECT  LOCAL  DEFAULT 97893 __func__.62732
...
```

If you look closely, the original vmlinux has the following two symbols:

```
 30010: ffffffff81361660  1383 FUNC    LOCAL  DEFAULT    1 format_corename.isra
 30011: ffffffff81361bd0    36 FUNC    LOCAL  DEFAULT    1 kmalloc_array.constp
```

While the built one does not! There are missing symbols in our freshly built vmlinux binaries. This is likely down to the “ISRA” optimisation round which GCC does. Maybe compiler flags are slightly different between builds. I am not sure. All I do know, is that this patch has problems.

Limitations in Livepatch

As we can see, there are some real limitations to which patches are suitable for Livepatch. This is probably the biggest reason why Livepatches are reserved for security fixes only, since most normal fixes won’t work.

The best cheat sheet for what patches work is the Patch Author Guide in the kpatch repository.

As soon as I can fix a real bug with Livepatch, I will write a follow up blogpost.

Installing and Configuring Livepatch on Ubuntu

Interested in using Livepatch in your production environment, but don't want to navigate all the complexity behind researching compatible patches, writing or generating Livepatch modules, testing for regressions or scaling deployment?

Well, you can use the Canonical Livepatch Service.

The Canonical Livepatch Service is easy to set up, and automatically delivers critical security fixes to your machines. These Livepatches have been thoroughly tested and are safe to use.

You can find a list of supported distribution releases and kernel versions on the Livepatch Wiki page.

The rule of thumb is that Livepatch is available for LTS GA kernels, and HWE kernels which are from the next LTS GA kernel.

So for example, 4.4 GA kernel on Xenial, or the 4.15 HWE kernel on xenial, since it was Bionic's GA kernel. Bionic will have 4.15 and soon, the 5.4 HWE kernel from Focal.

The Canonical Livepatch service is pretty easy to set up. All you need to do is:

1. Visit the Canonical Livepatch Portal to generate your API key.
2. Install the Livepatch system daemon with `$ sudo snap install canonical-livepatch`
3. Setup Livepatch with the API key: `$ sudo canonical-livepatch enable <TOKEN>`

You can try Livepatch for free for up to 3 machines, which is pretty neat if you want to use it on your own personal PC or server. If you need to scale for your production environment, then you can sign up for Ubuntu Advantage which includes the Canonical Livepatch Service.

The Datasheet covers any more questions you might have, such as on-premise availability or pricing.

So how do we tell if the Canonical Livepatch Service is working? Well, you can run:

```
$ canonical-livepatch status
last check: 1 minute ago
kernel: 4.4.0-168.197-generic
server check-in: succeeded
```

```
patch state: ✓ all applicable livepatch modules inserted
patch version: 65.1
```

We can also check dmesg, to see if the module has been inserted correctly:

```
[  234.112955] lkp_Ubuntu_4_4_0_168_197_generic_65: loading out-of-tree module
[  234.113077] lkp_Ubuntu_4_4_0_168_197_generic_65: module verification failed:
[  237.331850] livepatch: tainting kernel with TAINT_LIVEPATCH
[  237.331852] livepatch: enabling patch 'lkp_Ubuntu_4_4_0_168_197_generic_65'
```

We can see that we are running patch version 65.1. What does that mean? How do we see what is in each patch?

Well, you can sign up for the Ubuntu Security Announce mailing list. All new Livepatches are announced here, under `[LSN-VERSION]` tags. For example, the patch we just installed above is documented here:

[LSN-0065-1] Linux kernel vulnerability

Otherwise you can also browse the source code repositories.

- Xenial Livepatch Source Code
- Bionic Livepatch Source Code

If we have a look at the Xenial 65.1 patch for 4.4.0-168-generic, we have vmx fixes, mwifiex wifi driver fixes, btrfs fixes, and i915 graphics fixes. We can also see that they are built with `kpatch-build`: Makefile for Xenial 65.1 patch.

Most users probably aren't interested in what are in their Livepatches, but if you are interested, feel free to review.

Conclusion

Well, there we have it. We looked into how Livepatch works at a semi-technical level, we implemented a few Livepatches of our own and got them working.

It's a pity that I haven't managed to make a Livepatch to fix a real bug just yet, since I keep selecting fixes which aren't compatible, but as soon as I find one which is, I will write another blog post about it.

We also had a look at the Canonical Livepatch Service, and I was pretty happy with how easy it is to operate, compared to the endless trouble of making these modules yourself.

I think Livepatch is a very cool kernel technology, so keep an eye out on future blog posts where I delve into it some more.

I hope you enjoyed the read, and as always, feel free to contact me.

Matthew Ruffell

Related Posts

Debugging a Zero Page Reference Counter Overflow on the Ubuntu 4.15 Kernel 02 Sep 2020

Deploying an OpenStack Cluster in Ubuntu 19.10 13 Feb 2020

Analysis of an Out Of Memory Kernel Bug in the Ubuntu 4.15 Kernel 13 Dec 2019